International Journal of High Performance Computing Applications

A Composition Environment for MPI Programs

Narayan Desai, Ewing Lusk and Rick Bradshaw

The online version of this article can be found at:
http://hpc.sagepub.com/cgi/content/abstract/21/2/166

Published by:
§SAGE
http://www.sagepublications.com

Additional services and information for *International Journal of High Performance Computing Applications* can be found at:

Email Alerts: http://hpc.sagepub.com/cgi/alerts

Subscriptions: http://hpc.sagepub.com/subscriptions

Reprints: http://www.sagepub.com/journalsReprints.nav

Permissions: http://www.sagepub.co.uk/journalsPermissions.nav

Citations http://hpc.sagepub.com/cgi/content/refs/21/2/166

A COMPOSITION ENVIRONMENT FOR MPI PROGRAMS

Narayan Desai
Ewing Lusk
Rick Bradshaw

MATHEMATICS AND COMPUTER SCIENCE DIVISION,
ARGONNE NATIONAL LABORATORY, ARGONNE, IL 60439
(DESAI@MCS.ANL.GOV)

Abstract

While MPI is the most common mechanism for expressing parallelism, MPI programs are not composable by using current MPI process managers or parallel shells. We introduce MPISH2, an MPI process manager analogous to serial Unix shells. It allows the composition of MPI and serial Unix utilities with one another to perform scalable tasks across large numbers of Unix clients. This paper discusses in detail issues of process management and parallel tool composition.

Key words: MPI, process management, parallel shell, command language

1 Introduction

The shell is the most familiar interface to Unix systems. In general, it is the first contact that users have with Unix systems. Its ubiquity makes it the dominant mechanism through which command execution occurs.

Unix shells provide a rich environment for task automation, exposing command exit codes, providing control flow constructs, and organizing disparate programs into complex command pipelines. Users are familiar with the decomposition of complex tasks into the invocation of single-function utilities using these mechanisms.

While MPI is not as ubiquitous as Unix shells, it is the dominant mechanism used to express parallelism in scalable applications. Many high-performance implementations of MPI exist; indeed, MPI is so pervasive that a good MPI implementation is frequently cited as one of the requirements for new large-scale computational science machines.

Unfortunately, process management systems that can start MPI programs have not provided or exposed sufficient information for their composition with their serial analogues or even with each other. To address this issue, we have implemented `MPISH2`, an MPI process manager that provides a user interface and composition capabilities nearly identical to the Bourne shell.

This paper focuses on the issues involved in process management and parallel tool use. We motivate the discussion with a series of use cases and examples. To set the context, we begin by describing how our work on MPISH2 fits into related efforts. Next, we discuss the design and implementation of `MPISH2`. We then demonstrate how program composition techniques can benefit the larger community of users of parallel system software and tools.

2 Related Work

Shells have long been a subject of interest in the Unix research community. Starting with the original shell included with early Unix systems (Thompson 1975), shells have been augmented into relatively full-featured programming languages, including data types (Korn, Northrup, and Korn 1996). Because of the familiarity of the shell interface to Unix users, many attempts have been made to present a shell-like interface for program execution on parallel systems.

Existing shells provide users with a large range of functionality. The Bourne shell (Bourne 1978) offers process management features, job control, interactive program execution, command pipelines, and control flow constructs. Using these facilities, users can automate complex interactions between executables in a robust and intuitive fashion.

As parallel systems have become more common, several attempts have been made to offer analogous capabilities for parallel execution of serial and parallel programs. *Parallel shells* provide an interface for the execution of the same program across multiple systems. The key goal of these tools is to enable scalable, uniform execution of the same programs across large numbers of clients. Parallel Distributed Shell, `pdsh` (http://www.llnl.gov/linux/pdsh/pdsh.html), is a parallel shell that uses an `rsh` or `ssh` client and a strided, parallel approach to run tasks in parallel across many systems. The C3 tools (Flannery et al. 2000) provide a similar execution mechanism that also runs tasks through `rsh`.

These tools do an admirable job of starting processes scalably; however, they do not expose any of the Unix process information needed to compose commands effectively. Discrete exit statuses are not returned for each process executed. All control flow statements are executed serially and independently. Command pipelines, while usable in some cases, are still fundamentally serial constructs. Most importantly, existing parallel shells do not support MPI process startup.

MPI process management has also been the subject of much work over the past several years. Historically, MPI startup mechanisms have scaled poorly and performed badly overall (Butler, Gropp, and Lusk 2000). Two systems have addressed these issues over the past few years. MPD (Butler et al. 2003) uses a group of daemons, arranged in a ring topology, to scalably start MPICH2 MPI processes. Yod (Brightwell and Fisk 2001) provides similar capabilities in the Cplant software stack.

Both of these mechanisms provide highly scalable process startup and management services needed to execute MPI processes, but neither provides sufficient information for use in shell-style programming. MPD provides access to all exit statuses and to standard I/O multiplexed into single streams. Yod provides similar access to standard I/O but fails to provide any access to return codes. Most importantly, each of these systems provides the access to the `exec` system call without adding a framework around it for program composition. This exclusion makes it impossible to compose several discrete tasks into a command pipeline, or any more complex task construct.

Our earlier attempt to solve this problem was `MPISH`. It had all of the startup functions required to natively execute MPICH2 programs; however, the specification of execution locations was insufficient for many complex tasks.

The Unix and parallel shell landscape includes all of the facilities needed to build complex compositions of parallel programs; however, no single tool includes all of these facilities. The ideal tool would include the flexibility and power of a serial shell with the ability to scalably execute and compose parallel programs.

The work we present here has been motivated largely by the gains in system software scalability afforded by the use of MPI in system tools (Desai et al. 2004, 2005; Ong, Lusk, and Gropp 2001). This approach also has proved positive in terms of overall performance gains. More surprisingly, tools implemented by using MPI-based scalable components have proved far easier to troubleshoot and debug than their *ad hoc* analogues. The need to execute large numbers of small scalable tools brings execution issues clearly into focus. As additional parallel system tools become available, the desirability of tool composition increases at least as fast.

3 Design

The design of `MPISH2` followed from the idea that users need not treat parallel programs differently from serial ones. With such a uniform execution interface, parallel (and hence scalable) reimplementations of serial utilities could be automatically used by existing scripts. At the same time, we wanted to take the best features from each of the previously mentioned shells. The goal was a system that provided the flexibility and power of the Unix shell, with the scalable startup features of parallel shells and the ability to run MPI processes directly.

We chose the Bourne shell (Bourne 1978) as the language basis for `MPISH2`. It has the benefit of being ubiquitous and well understood among Unix users. The control flow constructs available in the Bourne shell are fairly standard, including `while`, `if`, `for`, and `case`. Since these are the real workhorses of shell scripting, we attempted to keep their semantics as close as possible to the Bourne shell. However, enhancements were required in order to support startup of parallel processes. In this section, we discuss these enhancements, as well as the language employed by `MPISH2` and the use of the resulting tool.

3.1 Enabling Parallelism

Parallel process managers work in much the same way as serial process managers. They are responsible for post-fork/pre-exec process setup and the setup of standard I/O. The main difference between serial and parallel process managers is the need for parallel library bootstrapping. This bootstrapping consists of two main parts: the description of the parallel process topology and the communication setup.

Many process managers describe initial process topology at the time of parallel process startup. Typically, the topology specification consists of process count and some set of resources, usually a list of nodes on which the processes should be executed. This corresponds closely to the common arguments to `mpirun`. Alternatively, one

can use `mpiexec`, specified by the MPI standard (MPI Forum 1994), for supplying the same data. Whatever the input format, this information is used for the same purpose: the description of initial communicator, `MPI_COMM_WORLD`, for the new process. Each communicator has a specific size, and each component process has a specified rank in that communicator. This initial topology description is what differentiates one 32-node program from thirty-two 1-node programs.

To support MPI program execution, we introduce the notion of a parallel execution context. A parallel execution context describes the resulting `MPI_COMM_WORLD` communicator for all MPI programs executed. When `MPISH2` is started, the initial parallel program execution context is global; it corresponds to `MPI_COMM_WORLD` for the `MPISH2` processes, since they are also an MPI process. As the `MPISH2` process executes the input script, the parallel execution context can be split into non overlapping pieces by `MPISH2` control flow constructs. This process is discussed in detail in Section 3.2.

The second important aspect of parallel process startup is communication bootstrapping. For disparate processes to begin acting as a single parallel entity, communication must be established. This is accomplished in different ways with different parallel libraries. MPICH2 uses an interface called PMI, or Process Manager Interface, to provide this information to client programs. PMI takes the form of a distributed database, providing standard *put, get,* and *fence* operations. The client program is provided with connection information for its PMI instance and can use that data to connect to other processes. Because this mechanism is not mandated by the MPI specification, this technique is implementation-specific. Hence, it works only with MPICH2 programs.

3.2 The MPISH2 Input Language

When considering how to integrate MPI programs into a Unix environment, we gave highest priority to retaining standard Unix shell semantics. The intention was to allow currently existing scripts to use `MPISH2` without modification. As parallel versions of Unix utilities are written, these scripts can transparently begin using them, greatly improving the scalability of existing tasks and processes. Several of these parallel tools are described in Section 4.2.

The compatibility requirement motivated the use of a Bourne shell syntax. All of the semantics regarding serial command execution remain the same as their serial shell counterparts. The Unix parent/child process relationship provides the same capabilities and remains fundamentally a serial construct. Command pipelines, backticks, and exit statuses also remain the same as those in the serial Bourne shell. However, the semantics of the con-

trol-flow constructs needed augmentation to support parallel execution contexts.

As described in the previous section, `MPISH2` supports parallel execution of MPI programs by using a parallel execution context. This context describes the topology of `MPI_COMM_WORLD` for any MPI processes executed by `MPISH2`. The current state of the parallel execution context at any given point is maintained by `MPISH2`. For example, if the first line of a script runs a program, its parallel execution context will be global, and processes will run in a single large context across all locations where `MPISH2` processes are running. As the script executes and control flow statements are processed, the parallel execution context is split into smaller pieces and then rejoined when those scopes disappear. This behavior is analogous to the use of MPI_Comm_split in MPI programs.

- *if* performs a two-way split, corresponding to the truth value of the predicate. `MPISH2` ranks will be grouped with others in the same side of the branch into two parallel execution contexts corresponding to true and false. For example, when *if* is executed in an 8-process context with a predicate that evaluates to true on 2 nodes and false on the other 6 nodes, ranks evaluating to true are grouped into a parallel execution context of size 2. Similarly, the remaining nodes are grouped into a second parallel execution context of size 6. These contexts persist until the *if* statement is finished executing.
- *case* performs an *N*-way split, operating similarly to *if*.
- *while* creates an execution context corresponding to all ranks for which the condition evaluates as true. All programs run in each iteration are grouped according to this initial evaluation. The condition is evaluated at the start of each iteration on each rank, continuing until all ranks evaluate false.
- *for* has no effect on parallel execution context because it is not conditional. No automatic parallelizing is performed.

Each of these control flow statements results in a set of new parallel execution contexts for the duration of the control flow statement. The formulation of these semantics requires the use of an implicit barrier at the conclusion of control flow execution. This approach has the benefit of retaining the character of the serial Bourne shell. All other Bourne shell semantics remain identical to their serial analogues; in fact, for the degenerate case, `MPISH2` behavior is identical to a serial Bourne shell.

3.3 MPISH2: A Parallel Shell

The most important difference between a normal shell and `MPISH2` is that `MPISH2` is a parallel program, consisting of multiple communicating Unix programs. A script,

given to MPISH2, is executed by each of the MPISH2 processes concurrently. The MPISH2 processes communicate with each other (in a scalable fashion) using MPI. That is, MPISH2 is itself an MPI program. Therefore, MPISH2 must be started by the startup mechanism of the proper MPI implementation. We assume in this paper that mpiexec invokes this mechanism. Thus, a 100-process instance of MPISH2 is started by a command line something like the following:

```
$ mpiexec -n 100 mpish2
```

In a cluster environment, the specification of which nodes MPISH2 is run on depends on the particular MPI implementation being used. We have used MPICH2 (http://www.mcs.anl.gov/mpi/mpich2), but MPISH2—being an MPI program—can be run by using any MPI implementation. Note, however, that because of the nonstandard nature of MPI startup, programs started by MPISH2 must use MPICH2.

As described in the previous section, MPISH2 scripts are Bourne-shell scripts that are presented to the standard input of each MPISH2 process. MPISH2 must be parallel in order to properly provide all information about child processes. For example, using a traditional MPI process manager to run two parallel programs in a pipeline would look like the following:

```
$ mpiexec -np 10 prog1 | mpiexec -np 10 prog2
```

This command runs prog1 and sends the standard output of the first mpiexec to the second invocation of mpiexec. Handling of standard output is not specified by the MPI standard; however, many MPI process managers provide multiplexed standard output from all processes to the standard output of mpiexec. Likewise, mpiexec typically, though not universally, sends standard input of mpiexec to some number of the parallel process instances. This approach is suboptimal for scripting, as the results are dependent on the implementation of the MPI process manager. Moreover, the construction of pipelines is fundamentally nonscalable and inefficient in this approach, since all stdio data is collected by the process management system and then redistributed for each subsequent stage in the command pipeline.

Under MPISH2, a similar command is used, together with a process management system for MPISH2 startup.

```
$ mpiexec -np 10 mpish2
```

Once MPISH2 is running, a command pipeline can be executed by using the following script.

```
$ prog1 | prog2
```

This script is run by every MPISH2 instance, resulting in 10 instances of both prog1 and prog2, connected rankwise into a pipeline. That is, standard output produced by the rank 0 instance of prog1 is fed into the standard input of the rank 0 instance of prog2, and so forth. Additional utilities are provided, allowing interrank manipulation of I/O streams. These execution semantics provide more flexibility and scalability than those afforded by traditional parallel shells and MPI process management systems.

4 Implementation

In this section, we discuss our implementation of MPISH2, and present several utilities we have written to provide a full user environment.

4.1 Shell Modifications

The implementation of MPISH2 is based on a modified version of the Minix (Tannenbaum 1987) shell, included with Busybox (http://www.busybox.net). Three main modifications have been made to this shell, corresponding to the issues described in the previous section.

First, MPISH2 needs to be able to provide a discrete PMI instance for each (potentially) parallel child program executed. PMI is a distributed database, including *put, get,* and *fence* operations. A new PMI instance is initialized whenever a new process is forked. Each is initialized with an MPI communicator that describes the current execution context. During client execution, each client program can connect to this PMI instance via a socket and issue commands.

Many of the commands, like *put*, which stores a value in a distributed database, will be serviced locally; however, some, such as *get* or *fence*, may require communication with other parts of the same PMI instance. All communication operations are implemented by using MPI collective and asynchronous operations. *Fence* is implemented by using MPI_Barrier. The implementation of *get* is more complicated. When a PMI instance receives a *get* request, it checks whether the value is already stored locally. If it is, the request is immediately serviced. If not, a message is sent to the PMI instance with the next higher rank modulo communicator size. Each process also receives queries for unknown values asynchronously. If the local process has the value, it responds to the querier; otherwise, it forwards the request to the next rank in the PMI instance.

Each PMI instance in the same MPISH2 process uses a discrete communicator that has been MPI_Comm_dup'ed at initialization time. This allows largely simultaneous execution of multiple parallel client programs; the only blocking operation used in the MPI implementa-

tion of PMI is the barrier used in the PMI *fence* operation.

The second major modification is driven by the fact that MPISH2 is designed to run parallel programs—that is, a set of processes grouped into a single cohesive entity. To support this, we added a stack of parallel program execution contexts to MPISH2. When MPISH2 begins execution, its initial execution context corresponds to its MPI_COMM_WORLD. However, as the script executes, the parallel execution context is split and joined based on conditional logic. Nested control flow statements result in a deeper stack of execution contexts, each with a corresponding MPI communicator.

The third, and perhaps most complex, modification was to the control flow construct to manipulate the current parallel execution context. In a typical serial shell, control flow constructs use only return codes and have no side effects. In MPISH2, however, control flow constructs also affect the parallel execution context by calling MPI_Comm_Split after predicate execution. For example, in serial shells, the shell executes the *if* predicate and either the true or false branch depending on a zero or nonzero return code, respectively. MPISH2 executes the same operations but with the addition of a call to MPI_Comm_Split using zero/nonzero exit status. Other control flow constructs were similarly modified.

The main complexity of implementing these control flow changes centered on *while*. Extending *while* to support parallel execution contexts required the addition of a parallel notion of while loop status. Normally, once the *while* predicate evaluates to false, the loop is complete. In an MPISH2 *while* loop, each rank needs to continue executing the loop predicate until all ranks evaluate to false.

None of these modifications proved complicated, and the overall semantics of the MPISH2 remains close to the semantics of the Bourne shell. At the same time, these modifications provide a wealth of new capabilities to Unix users.

4.2 Parallel Utilities

A parallel execution environment is not really complete without a set of parallel programs useful for writing basic programs. These programs are analogous to test or wc for serial shells. We have implemented a variety of small utilities, suffixed with the .mpi extension, to address this issue. The first set is a series of parallel predicates, suitable for use in control flow constructs. The following is a list of basic parallel predicates, with a short description of each.

- rank.mpi displays the process's rank in the current execution context.

- size.mpi displays the size of the current execution context.
- once.mpi exits with a return code of 0 once per physical node present.
- zoom.mpi provides access to scalable numeric reductions for the provided argument. The predicate use of this utility returns 0 when the argument falls within one standard deviation of the mean of all values.

Another group of utilities is used to move data between ranks in parallel pipeline operations.

- pflatten.mpi sends all stdout streams to process 0.
- ptee.mpi forwards stdin from process 0 to all processes. It functions like a parallel version of tee.
- pcoalesce.mpi coalesces stdout from all nodes, producing hostname delimited lines on processor 0.
- bcast.mpi broadcasts the data from one process, specified as an argument, to all other processes. This data is reproduced on stdout.

The final group of utilities comprises parallel analogues to serial utilities. These utilities provide the most promise for new types of functionality, as these MPI utilities provide the ability to use client systems as a broadcast tree and collective analysis of data.

- stagein.mpi downloads a file from an http server and broadcasts to all nodes, eventually writing it to disk on each.
- stageout.mpi uploads files, tagged with rank, to the fileserver from all clients.
- rsync.mpi synchronizes files from process 0 to all other processes. This program can handle all regular and special files.
- time.mpi times the execution of a parallel program, producing a single walltime result.

Each of these programs is a simple MPI program. Nothing special is required to write a utility, so users can easily write custom, scalable MPISH2 utilities.

5 Usage Examples

MPISH2 is useful across the same broad range of problems as are standard shell scripts, with the added ability to run concurrent, parallel programs. It can easily be used for tasks ranging from the most trivial to those that can strongly benefit from access to parallelism and scalable tools. In this section, we illustrate the most interesting language features and use cases for MPISH2. We begin by providing simple cases that show use of these features in isolation, and then build up to several more complex examples that demonstrate the flexibility and elegance of this approach.

5.1 Language Features

This first example demonstrates how conditional expressions interact with the parallel execution context.

5.1.1 Parallel execution context manipulation
The following script splits the current execution context into two: one consisting of up to the first four ranks, and another consisting of the remainder. Both branches run a hello world program demonstrating the size and details of each parallel process.

```
#!/usr/bin/env mpish2

if [ `rank.mpi` −lt 4 ] ; then
    hello.mpi
else
    hello.mpi
fi
```

Note that each of these branches can be further subdivided by running a second conditional inside.

5.1.2 Workload distribution among clients
The second example begins a command pipeline that performs a file listing on rank 0, broadcasts this listing to all clients, performs a local ls -l on each client, performs a local filter to exclude nonzero-length files, and coalesces the results to rank 0.

```
#!/usr/bin/env mpish2

(test `rank.mpi` −eq 0 && find /path −type f ) | \
    ptee.mpi | xargs ls −l | awk '{if ($5 == 0) print $0}' | \
    pcoalesce.mpi
```

Performing these same operations with traditional parallel shells would be dramatically less scalable, as all output processing would be performed on the head node. Hence, the complete results of all commands would need to be transmitted to the head node. Also, all processing of this output would be performed serially on this single node. This example demonstrates the ability to run different programs across ranks during the course of a single command pipeline.

5.1.3 Scalable utility replacement
The final case demonstrates the use of a scalable replacement for a standard Unix program, rsync. A simple invocation of rsync can cause substantial problems on a scalable resource with a limited fileserver infrastructure. Each client will individually download data from the server in a point-to-point fashion.

```
#!/usr/bin/env mpish2

rsync.mpi −av /src /dst
```

When run under MPISH2, this script is able to call a scalable replacement for rsync that performs an internal broadcast of file metadata and contents. This has the useful behavior of providing a constant load on the file service infrastructure, regardless of client scale; all per-client scaling is performed on the clients. This technique was demonstrated in an earlier paper (Desai et al. 2004) but has been made much more accessible by MPISH2.

5.2 Complex Examples

The following are more complicated examples of MPISH2 being used in common cluster tasks.

5.2.1 Job script
This example is a job script for a queueing system. This script runs the prologue, epilogue, and file staging commands once per physical node (hostname). Of these commands, the prologue and epilogue are serial, while the file staging commands are parallel. Once setup has completed, the user job is run (under the user's UID), and cleanup is performed. Not only can serial and parallel programs be interchanged, but standard shell scripting mechanisms (like the use of su) can also be used with parallel programs.

```
#!/usr/bin/env mpish2
user="${1}"
userscript="${2}"
indir="${3}"
outdir="${4}"

once.mpi
once=''${?}''
if [  ''${once}'' −eq 0 ] ; then
    # run the prologue once per node
    /usr/sbin/prologue
    if [ ! −z "${indir}"] ; then
        su "${user}" stagein.mpi "${indir}"
    fi
fi

su "${user}" mpish2 "${userscript}"


if [ ''${once}'' −eq 0 ] ; then
    if [ ! −z "${outdir}" ] ; then
        su "${user}" stageout.mpi "${outdir}"
    fi
    /usr/sbin/epilogue
fi
```

Several active execution contexts are used in this program. Two instances of a context containing each physical node are created by the script. The first is used for job setup (e.g. prologue and file staging), and the second is used for job cleanup. The user's job script is executed in the global execution context.

5.2.2 Benchmarking scripts

This example provides a basic illustration of concurrency. Benchmarking scripts are often implemented as a *for* loop that sequentially executes program runs with different sizes, for example, a script such as the following.

```
#!/bin/sh

for i in 2 4 8 16 32; do
  time mpirun −np $i program
done
```

Such a script does a reasonable job of running benchmarks; however, numerous processor resources are wasted in the first few iterations of the loop if the full number of nodes is reserved for the full duration of the execution.

This process can be run far more efficiently if test cases are executed concurrently. First, the application is run on all nodes. Second, the nodes are grouped into partitions, each with a different power of two size, up to half the total number of nodes. Each of these partitions runs a different size test case concurrently. The following example is a concurrent benchmarking script. It is assumed that the script is run on the largest size being benchmarked, in this case 32 nodes.

```
#!/usr/bin/env mpish2
rank=`rank.mpi`

slot="0"
basenum="2"
count="1"

time.mpi −t "size=32" progname

while [ "$slot" −eq "0" ] ; do
    remainder=`expr "$rank" − "$basenum"`
    if [ "$remainder" −lt "$basenum" ] ; then
        slot="$count"
    else
        basenum=`expr "$basenum" "*" "2"`
        count=`expr $count + 1`
    fi
done
```

```
case $slot
    1)
     time.mpi −t "size=2" progname
     ;;
    2)
     time.mpi −t "size=4" progname
     ;;
    3)
     time.mpi −t "size=8" progname
     ;;
    4)
     time.mpi −t "size=16" progname
     ;;
esac
```

6 Conclusions and Further Work

We have presented MPISH2, a parallel process manager for MPI programs that provides an interface almost indistinguishable from the standard Unix Bourne shell. It enables the use of MPI in Unix environments in a seamless manner not previously possible. The addition of scalable utilities and simple, Bourne shell-style control to Unix environments enables a variety of system and user tasks to be implemented in a scalable and elegant fashion. Moreover, users can now distribute macroscopic tasks scalably across pools of clients and explicitly control how communication occurs in command pipelines.

MPISH2 creates a venue in which scalable Unix utilities can be used. The range of current utilities available is clearly insufficient for all of the possible use cases. Hence, much of the future work will consist of identifying tasks that would be better performed in parallel. Also, one current limitation of MPISH2 is that job control has not been parallelized; it is unclear whether such parallelization is needed, but the issue should be investigated.

Acknowledgments

Author Biographies

Narayan Desai received his B.A. in computer science from the University of Chicago. He is a system administrator and system software developer in the Mathematics and Computer Science Division of Argonne National Laboratory. His work mainly falls at the intersection of system software and system management.

Ewing "Rusty" Lusk received his B.A. in mathematics from Notre Dame in 1965 and Ph.D. from the University of Maryland in 1970. He was an assistant professor of mathematics, and associate and full professor of computer science, at Northern Illinois University before joining Argonne National Laboratory in 1982. He is an Argonne Distinguished Fellow and director of the Mathematics and Computer Science Division at Argonne. His work has been in automated reasoning and parallel computing, including the development of the MPI message-passing standard, on which he has written and lectured widely. He is also the co-developer of the MPICH implementation of MPI. He currently works on programming models and tools for parallel computing. He is the author of more than 100 research articles in mathematics, automated deduction, and parallel computing, together with a number of books.

Rick Bradshaw is a system administrator in the Mathematics and Computer Science Division of Argonne National Laboratory. He helps to maintain HPC resources, experimental computing resources, and general UNIX infrastructure. He can be reached at bradshaw@mcs.anl.gov.

References

Bourne, S. R. (1978). An introduction to the Unix shell, *Bell System Technical Journal*, **57**(2): 2797–2822.

Brightwell, R. and Fisk, L. A. (2001). Scalable parallel application launch on Cplant, in *Proceedings of Super Computing 2001*.

Butler, R., Desai, N., Lusk, A., and Lusk, E. (2003). The process management component of a scalable system software environment, in *Proceedings of the 5th IEEE International Conference on Cluster Computing (CLUSTER03)*, pp. 190–198, IEEE Computer Society.

Butler, R., Gropp, W., and Lusk, E. (2000). A scalable process-management environment for parallel programs, in J. Dongarra, P. Kacsuk, and N. Podhorszki, editors, *Recent advances in parallel virutal machine and message passing interface*, Lecture Notes in Computer Science, vol. 1908, pp. 168–175, Springer.

Desai, N., Bradshaw, R., Lusk, A., and Lusk, E. (2004). MPI cluster system software, in D. Kranzlmuller, P. Kacsuk, and J. Dongarra, editors, *Recent advances in parallel virutal machine and message passing interface*, Lecture Notes in Computer Science, vol. 3241, pp. 277–286, Springer.

Desai, N., Lusk, A., Bradshaw, R., and Lusk, E. (2005). MPISH: A parallel shell for MPI programs, in *Proceedings of the 1st Workshop on System Management Tools for Large-Scale Parallel Systems (IPDPS '05)*, Denver, Colorado.

Flannery, R., Geist,A., Luethke, B., and Scott, S. L. (2000). Cluster command & control (C3) tools suite, in *Proceedings of the 3rd Distributed and Parallel Systems Conference*, Kluwer Academic Publishers, Norwell, MA.

Korn, D. G., Northrup, C. J., and Korn, J. (1996). The new Korn shell, *The Linux Journal*, **27**, July.

MPI Forum. (1994). Message Passing Interface Forum: Document for a standard message-passing interface, Technical Report CS-93-214 (revised), University of Tennessee. Available on **netlib**.

Ong, E., Lusk, E., and Gropp, W. (2001). Scalable Unix commands for parallel processors: A high-performance implementation, in Y. Cotronis and J. Dongarra, editors, *Recent advances in parallel virtual machine and message passing interface*, Lecture Notes in Computer Science, vol. 2131, pp. 410–418, Springer.

Tannenbaum, A. (1987). *Operating systems, design and implementation*, Prentice Hall, Old Tappan, NJ.

Thompson, K. (1975). The Unix command language, in *Structured Programming*, pp. 375–384, Infotech Information Ltd.